ANL-88-3

Mathematics and Computer Science Division
Mathematics and Computer Science Division
Mathematics and Computer Science Division

# Portable Implementation of a Generic Exponential Function in ADA

by Ping Tak Peter Tang

19980819 129

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38

U01247

Printed in the United States of America
Available from
National Technical Information Service
U. S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche Copy: A01

Distribution Category:
Mathematics and Computers
(UC-32)

**ANL-88-3**

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

# PORTABLE IMPLEMENTATION OF A GENERIC EXPONENTIAL FUNCTION IN ADA

*Ping Tak Peter Tang*

Mathematics and Computer Science Division

February 1988

This work was supported by the Strategic Defense Initiative Organization, Office of the Secretary of Defense, under WPD B411.

# Contents

# Portable Implementation of a Generic Exponential Function in Ada

by

*Ping Tak Peter Tang*

## Abstract

By presenting a provably accurate implementation of the exponential function, we illustrate that an accurate and portable elementary-function library can be implemented in Ada.

## 1 Introduction

Since July of 1986, the SIGAda Numerics Working Group has been working on a specification for the elementary transcendental functions for Ada. The specification includes requirements on accuracies for the various functions. Our interest lies not only in formulating the specification, but also in demonstrating portable implementations that are reasonably accurate and efficient. One of the goals is to illustrate that the proposed specification is well formulated, by implementing an elementary-function library that is both portable and provably accurate to well within the requirements of the specification.

This is a report on the exponential function that we implemented in conformance with the specification dated December 6, 1987. (Although the specification is still evolving, we expect further changes to be minor.) The next two sections discuss two different problems related to the construction of portable generic libraries. Solutions to the problems are also presented. These solutions will be applicable not only to the exponential function but also to the other functions. Sections 4 and 5 present the algorithm and the implementation details for the exponential function. Section 6 analyzes the implementation and provides an error bound for the computed result. Section 7 presents some results obtained from a number of tests performed on the function. Finally, Section 8 discusses further work to be pursued. The Appendix lists the complete source code.

## 2 Generic Packages and Range Constraints

### 2.1 Problem

The proposed specification requires that the library of elementary functions be generic and provide an accuracy comparable to the base type of the generic actual type. What type, then, should be used inside the generic package for computations? The generic actual type is unsuitable because it may have a range constraint that can be violated during intermediate calculations in the package. This violation would then cause a constraint error to be raised even if the final result would have been perfectly acceptable, had the exception not occurred.

It is clear that a constraint-free type with a precision comparable to that of the base type of the generic actual type must be used within the generic package body. Let WORKING_FLOAT be such an ideal type, and let FLOAT_TYPE be the generic formal type. How can WORKING_FLOAT be declared? One may try declaring

```
type WORKING_FLOAT is digits FLOAT_TYPE'BASE'DIGITS.
```

Unfortunately, this does not work because FLOAT_TYPE'BASE'DIGITS is a nonstatic expression. A workable solution, but impractical for portable implementations, would be to perform a case analysis as follows:

```
case FLOAT_TYPE'BASE'DIGITS is
when 1 =>
   declare
      type WORKING_FLOAT is digits 1;
   begin
     -- complete calculation of exp in this case
     -- approximation accurate to at least 1 digit
   end;
when 2 =>
   declare
      type WORKING_FLOAT is digits 2;
   begin
      -- complete calculation of exp in this case
      -- approximation accurate to at least 2 digits
   end;
           ..................

when SYSTEM.MAX_DIGITS =>
   declare
      type WORKING_FLOAT is digits SYSTEM.MAX_DIGITS;
   begin
      -- complete calculation of exp in this case
```

```
         -- approximation accurate to at least
         -- SYSTEM.MAX_DIGITS digits
      end;
   end case;
```

For any portable implementation that intends to accommodate systems whose attribute SYSTEM.MAX_DIGITS is 15 or larger, the solution just proposed leads to a huge code.

## 2.2 Solution

As a practical solution, we condense the many cases as follows:

```
case FLOAT_TYPE'BASE'DIGITS is
when 1..6 =>
   declare
      type WORKING_FLOAT is digits
      (6+SYSTEM.MAX_DIGITS - abs(6-SYSTEM.MAX_DIGITS))/2;
   -- the expression above is MIN( 6, SYSTEM.MAX_DIGITS )
   begin
      -- complete calculation of exp in this case
      -- approximation accurate to at least 6 digits
   end;
when 7..15 =>
   declare
      type WORKING_FLOAT is digits
      (15+SYSTEM.MAX_DIGITS - abs(15-SYSTEM.MAX_DIGITS))/2;
   begin
      -- complete calculation of exp in this case
      -- approximation accurate to at least 15 digits
   end;
          ..................

when 27..33 =>
   declare
      type WORKING_FLOAT is digits
      (33+SYSTEM.MAX_DIGITS - abs(33-SYSTEM.MAX_DIGITS))/2;
   begin
      -- complete calculation of exp in this case
      -- approximation accurate to at least 33 digits
   end;
when others =>
   -- cannot handle this case
end case;
```

This method guarantees that

$$\texttt{WORKING\_FLOAT'DIGITS} \geq \texttt{FLOAT\_TYPE'BASE'DIGITS} \quad (1)$$

always. Furthermore, to avoid using a type that is unnecessarily more accurate, we note that equality for (1) holds whenever the right boundary of the case coincides with a predefined type of the machine on which the code runs. Thus, we have chosen the cases such that on the all the Ada systems that we have experience with,

$$\texttt{WORKING\_FLOAT'DIGITS} = \texttt{FLOAT\_TYPE'BASE'DIGITS} \quad (2)$$

for all possible `FLOAT_TYPE`s.

## 3 Accurate Implementation with Extra-Precise Arithmetic

A typical implementation of an elementary function involves three steps: argument reduction to a primary interval, approximation of the function on that primary interval, and reconstruction of the function value. In accurate implementations, it is standard practice to perform argument reduction and reconstruction in an extra-precise data type. When such a data type is unavailable, extra precision is simulated in software using the working-precise data type (cf. [3], [5]). Therefore, an elementary function package following that practice would try to exploit extra-precise data types whenever they are available, and resort to simulation when they are not.

In principle, a portable generic package is able to detect at elaboration time whether an extra-precise data type is available, and consequently can ensure that appropriate actions be taken when the function is invoked later. In practice, however, because of technicalities in constructing a portable generic package, such an approach will lead to a huge code jammed with several sets of (otherwise unnecessary) constants used for argument reduction and reconstruction and with many complicated branches.

Fortunately, there are two practical alternatives to the impractical approach, one slightly more efficient while the other noticeably more accurate. We have chosen the latter. In what follows, we describe and compare the two methods, examine the cost of the method that we adopted, and explain how to switch from the adopted method to the other.

### 3.1 Two Methods

In Section 2.2, we described how we could declare the type `WORKING_FLOAT` that corresponds to the same predefined floating type of the generic actual type. Thus, an acceptably accurate implementation can compute solely in `WORKING_FLOAT` and perform simulation of extra precision at critical places. There is an advantage as well as disadvantages to this method.

- Advantage:

  The cost is minimal. The implementation is no more expensive than an acceptably accurate implementation need be. The price of using simulation has to be paid even when the package is not required to be generic or portable.

- Disadvantages:

  1. This method leads to a large body of code. The reason is that the three steps in the implementation — argument reduction, approximation, and reconstruction — must be included in each of the different cases of `WORKING_FLOAT`s (cf. Section 2.2).
  2. When extra precision is available without simulation, this method is less accurate than that of employing the readily available higher-precise arithmetic. The reason is that, because of the high cost, extra-precision simulation is done only in a few of the many places where higher precision would enhance accuracy noticeably.

To overcome the disadvantages, we have taken an approach that uses unsimulated extra precision whenever it is available. Because of portability and genericity, the only convenient extra-precise type is `LONGEST_FLOAT`, the type with the maximum allowable number of digits. Moreover, because we cannot determine a priori whether `WORKING_FLOAT` leaves us with any extra-precise type, the implementation must simulate extra-precision operations as well. Let us consider the advantages and disadvantages of this approach.

- Advantages:

  1. The resulting code is compact. The reason is that the code for both argument reduction and reconstruction need appear only once. These two steps compute solely in `LONGEST_FLOAT` and work for all possible generic actual types.
  2. The accuracy is enhanced in general. When

     `LONGEST_FLOAT'DIGITS > WORKING_FLOAT'DIGITS,`

     the result obtained would be more accurate than that obtained from computations done solely in `WORKING_FLOAT`, even with the help of extra-precision simulation.

- Disadvantages:

  1. Work is duplicated in this approach. When

     `LONGEST_FLOAT'DIGITS > WORKING_FLOAT'DIGITS,`

     the simulation of extra precision is rendered redundant by the use of the type `LONGEST_FLOAT`.

2. The approach may be unaffordably inefficient. It is conceivable that in some systems, operations in LONGEST_FLOAT are extremely inefficient. For example, it is possible that operations on the H-format (113 significant bits) data types supported by the VMS operating systems may be implemented in software on some particular machines.

We have chosen the second approach because of the higher accuracy it offers and the compact code that results from it. Section 3.2 below shows that the disadvantages discussed above are insignificant in most cases; and Section 3.3 discusses how the first approach can be implemented by slight modification of our code (given in the Appendix).

## 3.2 Cost

How often is work being duplicated in our approach? On systems with only two predefined floating types, duplication occurs only half of the time. In those undesirable cases, the cost of the unnecessary effort is only approximately five multiplications in LONGEST_FLOAT. Moreover, operations in LONGEST_FLOAT are efficient whenever they are implemented in hardware. Consequently, on machines with only two predefined floating types, both supported in hardware, our implementation is justified. This applies to all but two Ada systems that we know of.

## 3.3 Alternative

On systems such as the IBM/370 or VAX under VMS, there are usually three predefined floating types. If calculations in the widest format are excessively expensive, implementors can easily incorporate the argument reduction and the reconstruction into the approximation step of the code (cf. the Appendix). By doing so, all calculations will be performed in the base type of the generic actual type.

# 4 Algorithm

The algorithm follows. Implementation details are given in the next section.

**Step 1.** Filter out the exceptional cases. When the magnitude of the input argument $X$ is so large that an accurate result cannot be represented in the underlying data type, the exception ARGUMENT_ERROR should be raised. There are other situations in which that exception should be raised, and they are stated precisely in the specification.

**Step 2.** Reduce the input argument $X$ to $[-\log 2/64, \log 2/64]$. Obtain integers $n$, $m$, and $j$ and machine numbers $R_1$ and $R_2$ such that (up to roundoff)

$$X = n \log 2/32 + (R_1 + R_2),$$

$|R_1 + R_2| \leq \log 2/64$. Furthermore,

$$n = 32 \cdot m + j, \quad j = 0, 1, \ldots, 31.$$

Because of rounding errors, $|R_1 + R_2|$ may exceed $\log 2/64$ by a few units of its last place, and the calculated value of $n$ may also differ by 1 from the integer closest to $32X/\log 2$. The analysis later on will show that implementation is still accurate despite these rounding errors.

**Step 3.** Approximate $\exp(R_1 + R_2) - 1$ by a polynomial $p(R_1 + R_2)$, where

$$p(t) = t + a_1 t^2 + a_2 t^3 + \cdots + a_n t^{n+1}.$$

**Step 4.** Reconstruct $\exp(X)$ by

$$\exp(X) = 2^m (2^{j/32} + 2^{j/32} p(R_1 + R_2)).$$

## 5 Implementation Details

### 5.1 Assumptions about Floating-Point Arithmetic

Ada demands that certain behavior of the underlying floating-point arithmetic be satisfied when the operands involved are safe numbers. For example, for safe numbers A and B, A − B, in the absence of underflow, has to be exact whenever cancellation occurs. The reason is that, because of cancellation, the difference is a safe number. The same inference, however, cannot be made if the numbers A and B are merely machine numbers.

In practice, implementations must be able to handle machine-number input and, ideally, provide an accuracy with respect to the machine precision that is in general higher than that of the safe numbers. Consequently, our implementation needs to make certain assumptions about the floating-point data types of the target machines. These assumptions are related to the exponent width, the arithmetic, and the radix.

The assumptions are as follows:

- *Exponent Width:* We assume that the number of bits in the exponent field never exceeds $L/3$, where $L$ is the actual number of binary bits in the mantissa of the machine. Otherwise, the accuracy of the final result would degrade as the magnitude of the input argument becomes large. This assumption is built into the number of bits of the value $\log 2/32$ we have stored in the program.

- *Arithmetic:* Let A and B be two machine numbers such that

  $$2\mathtt{B} \geq \mathtt{A} \geq \mathtt{B}.$$

  Thus, cancellation occurs in A − B.

On binary machines, we assume that the subtraction is exact whenever B has one (or more) trailing zero bit(s).

On nonbinary machines, we assume that A − B is exact. This assumption requires in particular that a guard digit be present in the subtraction hardware.

If this assumption on arithmetic is violated, our scheme for argument reduction may fail to be accurate.

- *Radix:* We assume that the radix is either 2 (binary) or 16 (hexadecimal). We made this assumption because on most binary and hexadecimal floating-point arithmetic that we know of, the previous assumption about arithmetic is satisfied.

All the Ada systems with which we have experience satisfy our assumptions. The following tabulates those machines[1] (cf. [2]).

| System | Data Type | Radix | 'digits | Mantissa Length (in bits) | Exponent Width (in bits) |
|---|---|---|---|---|---|
| IBM/ 370 | single | 16 | 6 | 24 | 7 |
| | double | 16 | 15 | 48 | 7 |
| | quad | 16 | 20 | 96 | 7 |
| CRAY-1 | single | 2 | 13 | 48 | 14 |
| | double | 2 | 27 | 98 | 14 |
| VAX/ VMS | single | 2 | 6 | 24 | 8 |
| | d-format | 2 | 9 | 56 | 8 |
| | g-format | 2 | 15 | 53 | 11 |
| | h-format | 2 | 33 | 113 | 15 |
| IEEE/ 754 | single | 2 | 6 | 24 | 8 |
| | double | 2 | 15 | 53 | 11 |

## 5.2 Implementation

The following notes correspond to the algorithm in the previous section. All computations are carried out in the order prescribed by the parentheses. In the following discussions, X is the input argument, FLOAT_TYPE is the generic formal type, and LONGEST_FLOAT is the type declared as digits SYSTEM.MAX_DIGITS.

**Step 1.** The exceptional cases are as follows:

- Raise ARGUMENT_ERROR if $|X| \geq$ LARGE_THRESHOLD, where

$$\text{LARGE\_THRESHOLD} := 2 * \text{FLOAT\_TYPE'SAFE\_EMAX} * \log 2.$$

[1]The 'digits attributes for IBM/370 quad precision and VAX d-formats are shorter than the mantissa can offer because the exponent ranges of those data types are limited.

Note that for $|\mathtt{X}| \geq$ LARGE_THRESHOLD,

$$e^{\mathtt{X}} \geq 2 \cdot \mathtt{FLOAT\_TYPE'SAFE\_LARGE},$$

or

$$e^{\mathtt{X}} \leq \frac{1}{2} \cdot \mathtt{FLOAT\_TYPE'SAFE\_SMALL}.$$

Thus, some arguments that are not filtered out here may still warrant an exception. The computations in the next three steps will be able to handle those situations (cf. exceptional handling in the code).

- Return $1.0 + \mathtt{X}$ if $|\mathtt{X}| \leq$ SMALL_THRESHOLD, where

$$\mathtt{SMALL\_THRESHOLD} := \mathtt{FLOAT\_TYPE'BASE'EPSILON}.$$

**Step 2.** Let Y := LONGEST_FLOAT(X) be the input argument converted to LONGEST_FLOAT. To perform the argument reduction accurately, do the following.

- First, calculate N, $\mathtt{N}_1$, and $\mathtt{N}_2$ as follows:

```
N := LONGEST_INTEGER(Y * INV_L);
-- see explanation below on LONGEST_INTEGER
if |N| ≥ $2^8$ then
    $N_2$ := N mod $2^6$;
    --$N_2 = 0, 1, \ldots, 2^6 - 1$
    $N_1$ := N − $N_2$;
otherwise,
    $N_2$ := 0;
    $N_1$ := N.
```

INV_L is the value $32/\log 2$ represented in LONGEST_FLOAT. The declaration

```
type LONGEST_INTEGER is range
                SYSTEM.MIN_INT..SYSTEM.MAX_INT;
```

is meant to provide an integer type that would accommodate all the possible round-to-integer values of Y * INV_L. A better way, however, is to use the proposed primitive function ([6]) REAL_INT that returns in floating type the integral value closest to a given real number.

- The reduced argument is represented in two LONGEST_FLOAT variables $\mathtt{R}_1$ and $\mathtt{R}_2$. The idea is to represent $\log 2/32$ to extra precision by two LONGEST_FLOAT numbers $\mathtt{L}_1$ and $\mathtt{L}_2$. The values of $\mathtt{L}_1$ and $\mathtt{L}_2$ are chosen at run time in such a way that $\mathtt{L}_1 + \mathtt{L}_2$ represents $\log 2/32$ to a sufficient number of bits more than the mantissa length of LONGEST_FLOAT. After $\mathtt{L}_1$ and $\mathtt{L}_2$ are chosen, compute $\mathtt{R}_1$ and $\mathtt{R}_2$ as follows:

```
TMP := $N_1 * L_1$;
if |Y| ≥ |TMP| then
```

```
        $R_1 := Y - TMP$;
    otherwise,
        $R_1 := (Y - TMP/2) - TMP/2$;
    If $N_2 \neq 0$, $R_1 := R_1 - N_2 * L_1$;
    $R_2 := -N * L_2$.
```

- Finally, calculate M and J by

$$J := N \bmod 32; \quad M := (N - J)/32.$$

**Step 3.** Let WORKING_FLOAT be the type described in Section 2. The approximation polynomial is computed by a standard recurrence in WORKING_FLOAT as follows:

$$\begin{aligned} R &:= \mathtt{WORKING\_FLOAT}(R_1 + R_2); \\ \mathtt{POLY} &:= R * R * (A_1 + R * (A_2 + R * (\ldots + R * A_n) \ldots)); \\ Q &:= R_1 + (R_2 + \mathtt{LONGEST\_FLOAT}(\mathtt{POLY})). \end{aligned}$$

The coefficients are obtained from a Remez algorithm.

**Step 4.** Each of the values $2^{j/32}$, $j = 0, 1, \ldots, 31$, is calculated beforehand and represented by two LONGEST_FLOAT numbers

$$\mathtt{TWO\_TO\_J\_BY\_32(J,LEAD)} \quad \text{and} \quad \mathtt{TWO\_TO\_J\_BY\_32(J,TRAIL)}.$$

The leading part has thirteen significant bits, and the trailing part has full precision. Thus the sum of the two represents $2^{j/32}$ to roughly thirteen more bits than LONGEST_FLOAT has in its mantissa. The reconstruction is as follows:

$$\begin{aligned} F &:= \mathtt{TWO\_TO\_J\_BY\_32(J,LEAD)} + \mathtt{TWO\_TO\_J\_BY\_32(J,TRAIL)} \\ Q_1 &:= \mathtt{TWO\_TO\_J\_BY\_32(J,LEAD)} \\ Q_2 &:= \mathtt{TWO\_TO\_J\_BY\_32(J,TRAIL)} + F * Q \\ \mathtt{EXP} &:= 2^M * (Q_1 + Q_2) \quad \text{for binary machines} \\ \mathtt{EXP} &:= 2^M * Q_1 + 2^M * Q_2 \quad \text{for hexadecimal machines} \end{aligned}$$

## 6 Error Analysis with Ada's Model

The proposed specification for the exponential function EXP requires that

$$\left| \frac{\mathtt{EXP(X)} - e^X}{e^X} \right| \leq 4 * \mathtt{FLOAT\_TYPE'BASE'EPSILON}$$

for most input values X. The analysis to follow will show that with moderate assumptions about the underlying floating-point arithmetic, the accuracy requirement is easily achieved by our algorithm implemented as described. Since most systems perform arithmetic more accurately than prescribed by Ada's model, our analysis will inevitably be pessimistic. In Section 6.6, we will discuss accuracy in terms of machine precision.

Our goal in the analysis is to estimate, in terms of

`FLOAT_TYPE'BASE'EPSILON`,

the final error in the implemented function. It is obvious that our implementation is least accurate when

`FLOAT_TYPE'BASE'EPSILON = LONGEST_FLOAT'EPSILON`.

Thus, we will analyze this case only. Throughout the analysis, we will use $\epsilon$ and `radix` to denote `'EPSILON` and `'MACHINE_RADIX` of `LONGEST_FLOAT`, respectively, and "exponent width" to denote the number of binary bits in the exponent field of its predefined floating type. We find the following notation useful in error analysis:

- Typefaced letters, `X`, `Y`, `P`, `Q`, etc., denote real numbers that are representable exactly in the machine.
- Angle brackets $\langle \cdots \rangle$ denote the value of a real number "$\cdots$" rounded to machine precision. Thus, executing the statement

  $$\mathtt{A} := \mathtt{B} * \mathtt{C}$$

  in machine precision gives the value

  $$\mathtt{A} = \langle \mathtt{B} \cdot \mathtt{C} \rangle.$$

- $\epsilon$ denotes the value `FLOAT_TYPE'BASE'EPSILON`.
- Let $x$ be a real number. We define $\xi(x)$ to be the difference between the value of $x$ when rounded to working precision and $x$ itself, thus:

  $$\xi(x) := \langle x \rangle - x.$$

- If one uses $\langle \cdot \rangle$ and $\xi(\cdot)$, the relationship

  $$\langle \mathtt{A} \; op \; \mathtt{B} \rangle = \mathtt{A} \; op \; \mathtt{B} + \xi(\mathtt{A} \; op \; \mathtt{B})$$

  holds for any machine-precision values `A` and `B` and for each of the four basic operations $+$, $-$, $\cdot$, and $/$.

Given safe numbers A and B, and *op* in $\{+,-,\cdot,/\}$, let $k$ be the unique integer such that

$$2^k \leq |\mathtt{A}\ op\ \mathtt{B}| < 2^{k+1}.$$

Then, with the notation just introduced, the Ada model guarantees that

$$|\xi(\mathtt{A}\ op\ \mathtt{B})| \leq 2^k\,\epsilon.$$

We are now ready to perform the analysis.

## 6.1 Classification of Errors

We can classify the errors in our algorithm into three categories:

- *Error in reduction.* The computed reduced argument $\mathtt{R}_1 + \mathtt{R}_2$ differs from the correct one $r$ defined by the equation

$$x = (32m + j)\log 2/32 \ +\ r.$$

- *Error in approximation.* The approximating polynomial or rational function $p(r)$ differs from $\exp(r) - 1$.

- *Rounding errors.* Errors will be committed as we compute $p(r)$ and the final reconstruction; such is the nature of finite-precision arithmetic on computers.

Our analysis treats each of the three categories of error independently before combining them.

## 6.2 Error in Reduction

Let $\mathtt{R}_1, \mathtt{R}_2, \mathtt{N}, \mathtt{N}_1, \mathtt{N}_2, \mathtt{M}$, and J be the values as obtained in step 2 of the implementation. We estimate the difference between the value $\mathtt{R}_1 + \mathtt{R}_2$ and the correct reduced argument $r$, where

$$r = \mathtt{Y} - \mathtt{N}\cdot\log 2/32.$$

Several observations:

- $\mathtt{L}_1, \mathtt{L}_2$ are so chosen (see the source code for details) that

$$|\mathtt{L}_1 + \mathtt{L}_2 - \log 2/32| < 2^{-10}\cdot 2^{-\text{exponent width}}\cdot\epsilon \quad\text{and}$$
$$|\mathtt{L}_2| < 2^{-9}\cdot 2^{-\text{exponent width}}.$$

- Both $\mathtt{N}_1\mathtt{L}_1$ and $\mathtt{N}_2\mathtt{L}_1$ are safe numbers with at least one trailing zero. This is because $\mathtt{L}_1$ is a safe number with at least nine trailing zeros and both $\mathtt{N}_1$ and $\mathtt{N}_2$ never exceed eight bits by design.

- Consider the calculation of $R_1$:

```
    $TMP := N_1 * L_1$;
    if $|Y| \geq |TMP|$ then
        $R_1 := Y - TMP$;
    otherwise,
        $R_1 := (Y - TMP/2) - TMP/2$;
    if $N_2 \neq 0$, $R_1 := R_1 - N_2 * L_1$.
```

  The crucial observation is that the calculations above are error free when performed on any hexadecimal machine with a guard digit for subtraction, and on any binary machines with or without a guard bit for subtraction. The reason is that both $N_1L_1$ and $N_2L_1$ have at least one trailing zero bit, and cancellation occurs in each of the subtractions above.

Using these observations, we can estimate the error in reduction as follows:

$$\begin{aligned} R_1 + R_2 &= Y - (N \cdot L_1 + \langle N \cdot L_2 \rangle) \\ &= Y - N(L_1 + L_2) + \xi(N \cdot L_2). \end{aligned}$$

Now, because

$$\begin{aligned} |N| &\leq |Y \cdot 32/\log 2| \\ &\leq |\text{LARGE\_THRESHOLD} \cdot 32/\log 2| \\ &\leq |64 \cdot 2^{\text{exponent width}-1} \cdot \log_2(\text{radix})| \\ &\leq \log_2(\text{radix}) \cdot 2^{\text{exponent width}+5}, \end{aligned}$$

therefore

$$|N \cdot L_2| < 2^{-6} \qquad \text{and} \qquad |\xi(N \cdot L_2)| \leq 2^{-7}\epsilon.$$

Consequently,

$$\begin{aligned} |(R_1 + R_2) - (Y - N \cdot \log 2/32)| &\leq N \cdot |L_1 + L_2 - \log 2/32| + |\xi(N \cdot L_2)| \\ &\leq 2^{-7}\epsilon + 2^{-7}\epsilon \\ &\leq 2^{-6}\epsilon. \end{aligned}$$

## 6.3 Error in Approximation

We estimate the difference between the transcendental function $e^t - 1$ and the approximating polynomial

$$p(t) = t + A_1 t^2 + \cdots + A_n t^{n+1}$$

for $t \in [-\log 2/64, \log 2/64]$. The estimation is done by locating numerically all the extreme points of $e^t - 1 - p(t)$ in the interval $[-0.010831, 0.010831]$ (slightly wider than

$[-\log 2/64, \log 2/64]$). In our code, five different polynomials are used for different ranges of `FLOAT_TYPE'DIGITS`. In each of those ranges, we find that

$$|e^t - 1 - p(t)| < 2^{-6}\epsilon$$

for all $t \in [-0.010831, 0.010831]$, where $\epsilon$ corresponds to the maximum number of digits in that particular range.

## 6.4 Rounding Errors

Here we are concerned with the difference between the value EXP obtained by executing

$$\begin{aligned}
\mathtt{R} &:= \mathtt{R}_1 + \mathtt{R}_2 \\
\mathtt{POLY} &:= \mathtt{R} * \mathtt{R} * (\mathtt{A}_1 + \mathtt{R} * (\mathtt{A}_2 + \mathtt{R} * (\ldots + \mathtt{R} * \mathtt{A}_n)\ldots)) \\
\mathtt{Q} &:= \mathtt{R}_1 + (\mathtt{R}_2 + \mathtt{LONGEST\_FLOAT(POLY)}) \\
\mathtt{F} &:= \mathtt{TWO\_TO\_J\_BY\_32(J,LEAD)} + \mathtt{TWO\_TO\_J\_BY\_32(J,TRAIL)} \\
\mathtt{Q}_1 &:= \mathtt{TWO\_TO\_J\_BY\_32(J,LEAD)} \\
\mathtt{Q}_2 &:= \mathtt{TWO\_TO\_J\_BY\_32(J,TRAIL)} + \mathtt{F} * \mathtt{Q} \\
\mathtt{EXP} &:= 2^{\mathtt{M}} * (\mathtt{Q}_1 + \mathtt{Q}_2) \quad \text{for binary machines} \\
\mathtt{EXP} &:= 2^{\mathtt{M}} * \mathtt{Q}_1 + 2^{\mathtt{M}} * \mathtt{Q}_2 \quad \text{for hexadecimal machines}
\end{aligned}$$

and the value we would have obtained had all the preceding calculations been error free.

Three observations simplify our analysis. First, on binary machines, the execution of

$$2^{\mathtt{M}} * (\mathtt{Q}_1 + \mathtt{Q}_2)$$

and

$$2^{\mathtt{M}} * \mathtt{Q}_1 + 2^{\mathtt{M}} * \mathtt{Q}_2 \quad .$$

yields identical results. Second, the magnitude of POLY is at most $\frac{1}{2}(\log 2/64)^2$, which is less that $2^{-13}$. Thus the rounding errors accumulated in POLY are practically zero. Third, $2^{\mathtt{M}} * \mathtt{Q}_1$ is exact because $\mathtt{Q}_1$ is a safe number.

To shorten the exposition that follows, we use $\mathtt{S}_1$ and $\mathtt{S}_2$ to denote

$$\mathtt{TWO\_TO\_J\_BY\_32(J,LEAD)} \quad \text{and} \quad \mathtt{TWO\_TO\_J\_BY\_32(J,TRAIL)},$$

respectively. We are now ready to begin.

Using the observations and the notation, we are going to estimate the difference between

$$\langle 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}} \cdot \langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\rangle\rangle$$

and

$$2^{\mathtt{M}} \cdot \mathtt{Q}_1 + (2^{\mathtt{M}} \cdot (\mathtt{S}_2 + ((\mathtt{S}_1 + \mathtt{S}_2) \cdot (\mathtt{R}_1 + (\mathtt{R}_2 + \mathtt{POLY})))))).$$

To obtain a good estimate, we must give a careful account for each deviation of our computed value from the ideal one. We use $E_0$ to denote the ideal result. $E_1$ denotes the first corrupted result, $E_2$ the second, and so on. $E_7$ is the final computed result, and the rounding error is simply the difference $E_0 - E_7$.

$$\begin{aligned}
E_0 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + (2^{\mathtt{M}} \cdot (\mathtt{S}_2 + ((\mathtt{S}_1 + \mathtt{S}_2) \cdot (\mathtt{R}_1 + (\mathtt{R}_2 + \mathtt{POLY})))))\\
E_1 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + 2^{\mathtt{M}} \cdot (\mathtt{S}_2 + (\mathtt{S}_1 + \mathtt{S}_2) \cdot (\mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle))\\
E_2 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + 2^{\mathtt{M}} \cdot (\mathtt{S}_2 + (\mathtt{S}_1 + \mathtt{S}_2) \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle)\\
E_3 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + 2^{\mathtt{M}} \cdot (\mathtt{S}_2 + \langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle)\\
E_4 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + 2^{\mathtt{M}} \cdot (\mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle)\\
E_5 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + 2^{\mathtt{M}} \cdot \langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\\
E_6 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}} \cdot \langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\rangle\\
E_7 &:= \langle 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}} \cdot \langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\rangle\rangle
\end{aligned}$$

We also name the following values by $F_1, F_2, \ldots, F_7$ because these values arise often in what follows.

$$\begin{aligned}
F_1 &:= \mathtt{R}_2 + \mathtt{POLY}\\
F_2 &:= \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\\
F_3 &:= \mathtt{S}_1 + \mathtt{S}_2\\
F_4 &:= \langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\\
F_5 &:= \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\\
F_6 &:= 2^{\mathtt{M}} \cdot \langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\\
F_7 &:= 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}} \cdot \langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2\rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\rangle
\end{aligned}$$

Now the estimates:

$$|\text{rounding errors}| = |E_0 - E_7| \leq \sum_{i=1}^{7} |E_{i-1} - E_i|,$$

and

$$\begin{aligned}
|E_0 - E_1| &= 2^M |\mathtt{S}_1 + \mathtt{S}_2| \cdot |(\mathtt{R}_2 + \mathtt{POLY}) - \langle \mathtt{R}_2 + \mathtt{POLY}\rangle|\\
&= 2^M |F_3| \cdot |\xi(F_1)|,\\
|E_1 - E_2| &= 2^M |F_3| \cdot |\xi(F_2)|,\\
|E_2 - E_3| &= 2^M |\langle F_2\rangle| \cdot |\xi(F_3)|,\\
|E_3 - E_4| &= 2^M |\xi(F_4)|,\\
|E_4 - E_5| &= 2^M |\xi(F_5)|,\\
|E_5 - E_6| &= |\xi(F_6)|,\\
|E_6 - E_7| &= |\xi(F_7)|.
\end{aligned}$$

To get an estimate of $|\xi(F_j)|$ for $j = 1, \ldots, 7$, we need know only the rightmost binary intervals in which the various $|F_j|$'s may lie. Note that each of the $F_j$'s is the computed result of some value whose range is known. Consequently, unless the largest magnitude achieved by those values lies very close to a power of 2, the rightmost binary intervals in which those values may lie are the binary intervals we seek. We tabulate the results below.

| Value | Range | Conclusion Drawn |
|---|---|---|
| $\lvert \mathtt{R}_2 \rvert$ | $[0, 2^{-10.78}]$ | $\lvert\xi(F_1)\rvert \leq 2^{-11}\epsilon$ |
| $\lvert p(r) \rvert$ | $[0, 2^{-6.52}]$ | $\lvert\xi(F_2)\rvert \leq 2^{-7}\epsilon$, $\lvert F_2\rvert \leq 2^{-6.5}$ |
| $\lvert 2^{j/32} \rvert$ | $[0, 2^{31/32}]$ | $\lvert\xi(F_3)\rvert = 0$ for $j = 0$; $\lvert\xi(F_3)\rvert \leq \epsilon$ otherwise. |
| $\lvert 2^{j/32} p(r) \rvert$ | $2^{j/32}[0, 2^{-6.52}]$ | $\lvert\xi(F_4)\rvert \leq 2^{-7}\epsilon$ for $j = 0$; $\lvert\xi(F_4)\rvert \leq 2^{-6}\epsilon$ otherwise. |
| $\lvert \mathtt{S}_2 + 2^{j/32} p(r) \rvert$ | $2^{j/32}[0, 2^{-6.49}]$ | $\lvert\xi(F_5)\rvert \leq 2^{-7}\epsilon$ for $j = 0$; $\lvert\xi(F_5)\rvert \leq 2^{-6}\epsilon$ otherwise. |
| $2^M \lvert \mathtt{S}_2 + 2^{j/32} p(r) \rvert$ | $2^M 2^{j/32}[0, 2^{-6.49}]$ | $\lvert\xi(F_6)\rvert \leq 2^M 2^{-7}\epsilon$ for $j = 0$; $\lvert\xi(F_6)\rvert \leq 2^M 2^{-6}\epsilon$ otherwise. |
| $\lvert 2^M 2^{j/32} e^r \rvert$ | $2^M 2^{j/32}[2^{-1/64}, 2^{1/64}]$ | $\lvert\xi(F_7)\rvert \leq 2^{M-1}\epsilon$ for $2^{j/32}e^r \leq 1$; $\lvert\xi(F_7)\rvert \leq 2^M\epsilon$ otherwise. |

Thus, when $j = 0$,

$$\begin{aligned} |\text{rounding error}| &\leq |\xi(F_7)| + 2^{M-1} \cdot \epsilon \cdot (2^{-10} + 2^{-6} + 0 + 2^{-6} + 2^{-6} + 2^{-6}), \\ &\leq |\xi(F_7)| + 2^{M-1} \cdot \epsilon \cdot 0.06348. \end{aligned}$$

When $j = 1$,

$$\begin{aligned} |\text{rounding error}| &\leq |\xi(F_7)| + 2^{M} \cdot \epsilon \cdot (2^{-10} + 2^{-6} + 2^{-6.5} + 2^{-6} + 2^{-6} + 2^{-6}), \\ &\leq |\xi(F_7)| + 2^{M} \cdot \epsilon \cdot 0.07453. \end{aligned}$$

Note also that

$$e^{\mathtt{Y}} = 2^M \cdot 2^{j/32} \cdot e^r,$$

and

$$|\xi(F_7)| \Big/ (2^M \cdot 2^{j/32} \cdot e^r) \leq \epsilon.$$

## 6.5 Overall Error

Finally, we estimate the overall relative error

$$\left| e^{\mathtt{Y}} - \langle 2^{\mathtt{M}} \cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}} \cdot \langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1 + \mathtt{S}_2 \rangle \cdot \langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY} \rangle\rangle\rangle\rangle\rangle\rangle \right| \Big/ e^{\mathtt{Y}}.$$

From the previous analysis,

$$
\begin{aligned}
&|\text{absolute error}| \\
&\quad = \left|2^{\mathtt{M}}2^{j/32}e^{r} - \langle 2^{\mathtt{M}}\cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}}\cdot\langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1+\mathtt{S}_2\rangle\cdot\langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\rangle\rangle\right| \\
&\quad \le 2^{\mathtt{M}}2^{j/32}\cdot|e^{r}-e^{\mathtt{R}_1+\mathtt{R}_2}| + 2^{\mathtt{M}}2^{j/32}\cdot|e^{\mathtt{R}_1+\mathtt{R}_2}-1-p(\mathtt{R}_1+\mathtt{R}_2)| \\
&\quad\quad + |2^{\mathtt{M}}2^{j/32}p(\mathtt{R}_1+\mathtt{R}_2) + 2^{\mathtt{M}}2^{j/32} \\
&\quad\quad\quad -\langle 2^{\mathtt{M}}\cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}}\cdot\langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1+\mathtt{S}_2\rangle\cdot\langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\rangle\rangle| \\
&\quad \le 2^{\mathtt{M}}2^{j/32}\left(1.01|r-(\mathtt{R}_1+\mathtt{R}_2)| + |e^{\mathtt{R}_1+\mathtt{R}_2}-1-p(\mathtt{R}_1+\mathtt{R}_2)|\right) \\
&\quad\quad + |2^{\mathtt{M}}2^{j/32}p(\mathtt{R}_1+\mathtt{R}_2) + 2^{\mathtt{M}}2^{j/32} \\
&\quad\quad\quad -\langle 2^{\mathtt{M}}\cdot \mathtt{Q}_1 + \langle 2^{\mathtt{M}}\cdot\langle \mathtt{S}_2 + \langle\langle \mathtt{S}_1+\mathtt{S}_2\rangle\cdot\langle \mathtt{R}_1 + \langle \mathtt{R}_2 + \mathtt{POLY}\rangle\rangle\rangle\rangle\rangle\rangle| \\
&\quad \le 2^{\mathtt{M}}2^{j/32}\,(1.01|\text{error in reduction}| + |\text{error in approximation}|) \\
&\quad\quad + |\text{rounding error}| \\
&\quad \le 2^{\mathtt{M}}2^{j/32}(1.01\cdot 2^{-6} + 2^{-6})\epsilon + |\text{rounding error}|.
\end{aligned}
$$

When $j = 0$, $e^{\mathtt{Y}} \ge 2^{\mathtt{M}-1}$ and

$$
\begin{aligned}
&|\text{relative error}| \\
&\quad \le 2\,(1.01\cdot 2^{-6} + 2^{-6})\,\epsilon + 0.06348\epsilon + \left(|\xi(F_7)|\;/\;e^{\mathtt{Y}}\right) \\
&\quad \le 1.13\epsilon.
\end{aligned}
$$

When $j \ge 1$, $e^{\mathtt{Y}} \ge 2^{\mathtt{M}}$ and

$$
\begin{aligned}
&|\text{relative error}| \\
&\quad \le 2^{31/32}\,(1.01\cdot 2^{-6} + 2^{-6})\,\epsilon + 0.07453\epsilon + \left(|\xi(F_7)|\;/\;e^{\mathtt{Y}}\right) \\
&\quad \le 1.14\epsilon.
\end{aligned}
$$

Thus, the relative error of the implementation stays well within the required threshold of $4\epsilon$.

### 6.6 Remarks

On all Ada systems that we have experience with, the implementation is actually capable of delivering comparable accuracy with respect to the precision offered by the underlying hardware. Moreover, on machines such as the VAX or those with floating-point arithmetic conforming to ANSI/IEEE Standard 754-1985, the previous analysis is pessimistic. In particular, a similar implementation that is tailored specifically to IEEE 754 arithmetic has been proved accurate to within 0.54 unit of last place. In a later paper, we will analyze in detail the accuracy of our implementation on the various machines we are interested in.

## 7 Test Results

The code as listed in the Appendix has been run on a Sequent VADS compiler version 5.41.6, an IBM PC/AT using the Meridian AdaVantage compiler version 2.0, and a VAX 8650 using DEC Ada version 1.4.

We have also tested the implementation on the Sequent and the IBM PC/AT using the ELEFUNT test transcribed into Ada by K. W. Dritz.

On the Sequent, there are two predefined floating-point types with 24 and 53 significant bits, `'digits` 6 and 15, respectively. Thus the accuracies offered by the two sets of safe numbers are 21 and 51 bits, respectively; and those offered by the two machine formats are 24 and 53 bits, respectively. The ELEFUNT results are summarized as follows. (For a description of the test, see [3].)

| Generic Actual Type | Accuracy with respect to | Reported Loss of Binary Bits: Max. Relative Error | Reported Loss of Binary Bits: Root Mean Square |
|---|---|---|---|
| `'digits 6` | 21 bits | 0.00 | 0.00 |
| `'digits 6` | 24 bits | 0.99 | 0.00 |
| `'digits 15` | 51 bits | 0.00 | 0.00 |
| `'digits 15` | 53 bits | 0.99 | 0.00 |

On the IBM PC/AT with the Meridian AdaVantage, there is only one predefined floating-point type with 53 significant bits, `'digits` 15. Thus, the accuracies offered by the safe numbers and the machine format are 51 bits and 53 bits, respectively. The results are summarized as follows.

| Generic Actual Type | Accuracy with respect to | Reported Loss of Binary Bits: Max. Relative Error | Reported Loss of Binary Bits: Root Mean Square |
|---|---|---|---|
| `'digits 15` | 51 bits | 0.00 | 0.00 |
| `'digits 15` | 53 bits | 1.00 | 0.00 |

## 8 Conclusion and Future Work

We have shown that the environmental inquiries and other numerical features provided by Ada make portability and provability of some numerical software possible. With conscientious effort, a reasonably portable and accurate exponential function can be implemented.

Our experience with the sample implementation presented here strongly suggests that the following four projects are within reach. We are committed to the first two, and we hope that circumstances will allow us to pursue the others.

- **Specification of Elementary Functions:** The sample implementation has provided us with valuable guidelines on proposing a specification for the elementary function library. We will continue to participate in the formulation of the specification.

- **Library of Elementary Functions:** A portable complete library of the twenty elementary functions ([7] and [6]) can be implemented by using strategies similar to those employed here. The only technical challenge we foresee now is an accurate reduction routine that finds the remainder of a machine number with respect to the transcendental number $\pi$.

- **Library of Primitive Functions:** We intend to construct a library of primitive functions similar to those proposed in [6]. Since the functions here are of a much lower level, a portable implementation may not be practical or even possible in some cases. Some of the elementary functions may be constructed on top of this basic library. (Our exponential function, though not dependent on this library, will benefit from it.)

- **Validation:** From our experience so far, a portable test suite seems to be implementable. The test suite we have in mind consists of two parts. The first part basically will be a transcription of the ELEFUNT test in [3]. The ELEFUNT test is adept in reporting possible mistakes and in estimating the accuracy of the function under test. The second part of the test will try to report the exact deviation of the function under test from the correct value. In the past, such a task has usually been performed only if higher-precision function values are available. With the numerical features of Ada — for example, accurate conversion of universal reals to model numbers, and table-driven techniques like those employed in our exponential function — we believe such a task can be accomplished portably even without an extra-precise function.

  A test suite as such should also be useful in validating libraries that claim conformance to the proposed specification.

## Acknowledgments

The bulk of the work related to implementation was done by the author. Many of the ideas on table-driven techniques were developed while the author worked under the supervision of Professor W. Kahan of the University of California at Berkeley. Similar table-driven implementations can also be found in [1] and [5]. This work has benefited from valuable suggestions from W. J. Cody, K. W. Dritz, and B. T. Smith, who together with the author are also actively working on the specification. The author also thanks W. J. Cody for his many valuable suggestions concerning our implementation.

Different versions of the implementation were also tested by J. Squire at Westinghouse and G. Hodgson at NAG.

## Appendix

The following is the complete source program for the exponential program.

```
package MATHEMATICAL_EXCEPTIONS is

   ARGUMENT_ERROR : exception;

end MATHEMATICAL_EXCEPTIONS;


with MATHEMATICAL_EXCEPTIONS;

generic

   type FLOAT_TYPE is digits <>;

package GENERIC_ELEMENTARY_FUNCTIONS is

   function EXP( X : FLOAT_TYPE ) return FLOAT_TYPE;

   -- other functions to be added later

   ARGUMENT_ERROR : exception renames MATHEMATICAL_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_ELEMENTARY_FUNCTIONS;


with SYSTEM;
with TEXT_IO;       use TEXT_IO;

package body GENERIC_ELEMENTARY_FUNCTIONS is

   -- As of 2/4/88, this package contains only the exponential
   -- function. More functions will be added later.

   -- FLOAT_TYPE is the floating-point type with which the user
   -- instantiates this package. Computation in this type is
   -- avoided to insulate ourselves from any possible range
   -- constraints imported with the type.

   -- Two floating-point types are defined in this package body:

   -- LONGEST_FLOAT is the floating-point type having 'DIGITS equal to
   -- SYSTEM.MAX_DIGITS. This type is needed here to perform
   -- argument reductions and final reconstructions of elementary
   -- function values in the maximum precision available.
-- WORKING_FLOAT is the floating-point type in which the approximations
-- for elementary functions are carried out. This type is so defined
-- that
--        WORKING_FLOAT'DIGITS = FLOAT_TYPE'BASE'DIGITS
-- on all the Ada systems we have experience with.
-- However, there may be some (unknown to us) systems for which
--        WORKING_FLOAT'DIGITS > FLOAT_TYPE'BASE'DIGITS
-- Thus, type WORKING_FLOAT has at least the precision of the base
-- type of FLOAT_TYPE, and usually it does not have excess precision.


-- Assumptions:

-- This package body is portable to a particular implementation only
-- if the following assumptions are valid in that implementation:

-- (1)    6 <= SYSTEM.MAX_DIGITS <= 33
-- (2)    The following assumptions are made on floating-point
--        arithmetic:
--
--        (a) Radix: The radix will be either 2 or 16.
--        (b) Exponent Width: We assume that the number of bits in
--            the exponent field of the floating-point format never
--            exceeds L/3, where L is the actual number of bits
--            in the mantissa of the machine.
--        (c) Arithmetic: Let A and B be two machine numbers
--            such that 2B >= A >= B. Then, cancellation occurs in
--            A - B. On binary machines, we assume that the
--            subtraction is exact whenever B has one (or more)
--            trailing zero bit(s). On nonbinary machines, we
--            assume that A - B is exact.  This assumption requires
--            in particular that a guard digit be present in the
--            subtraction hardware for the nonbinary machines.

-- If the assumptions (1) and (2a) are invalid, the predefined
-- exception PROGRAM_ERROR will be raised.
-- Some compilers could detect at compile time that it
-- must always be raised at run time, thus calling attention to the
-- violation of the assumptions at compile time.


type    LONGEST_FLOAT is digits SYSTEM.MAX_DIGITS;
type    LONGEST_INTEGER is range SYSTEM.MIN_INT..SYSTEM.MAX_INT;
type    POSITION     is (LEAD, TRAIL);
subtype INDEX is LONGEST_INTEGER range 0..31;

-- The following tables of constants are needed by several of the elementary
-- functions.

-- TWO_TO_J_BY_32 is an array of 32 pairs of LONGEST_FLOAT numbers
-- representing 2**(j/32) for j = 0, 1, 2, ..., 31. Each such value
-- is represented by LEAD + TRAIL. The leading parts contain 13 bits
-- of information and are consequently model numbers as long as
-- SYSTEM.MAX_DIGITS is >= 4. The trailing parts contain roughly
-- LONGEST_FLOAT'MANTISSA bits of information, under the assumption
-- that SYSTEM.MAX_DIGITS is <= 35. So, when the assumptions are met,
-- LEAD + TRAIL represents 2**(j/32) to roughly 13 extra bits.

TWO_TO_J_BY_32 : constant array( INDEX, POSITION ) of LONGEST_FLOAT := (
   0 => (LEAD =>16#1.000#, TRAIL =>16#0.000000000000000000000000000000000#),
   1 => (LEAD =>16#1.059#, TRAIL =>16#0.000B0D31585743AE7C548EB68CA417FE5#),
   2 => (LEAD =>16#1.0B5#, TRAIL =>16#0.000586CF9890F6298B92B71842A983642#),
   3 => (LEAD =>16#1.113#, TRAIL =>16#0.00001D0125B50A4EBBF1AED9318CEAC5C#),
   4 => (LEAD =>16#1.172#, TRAIL =>16#0.000B83C7D517ADCDF7C8C50EB14A79203#),
   5 => (LEAD =>16#1.1D4#, TRAIL =>16#0.000873168B9AA7805B8028990F07A98B4#),
   6 => (LEAD =>16#1.238#, TRAIL =>16#0.0007A6E75623866C1FADB1C15CB593B03#),
   7 => (LEAD =>16#1.29E#, TRAIL =>16#0.0009DF51FDEE12C25D15F5A24AA3BCA88#),
   8 => (LEAD =>16#1.306#, TRAIL =>16#0.000FE0A31B7152DE8D5A46305C85EDECB#),
   9 => (LEAD =>16#1.371#, TRAIL =>16#0.000A7373AA9CAA7145502F4547987E3E1#),
  10 => (LEAD =>16#1.3DE#, TRAIL =>16#0.000A64C12342235B41223E13D773FBA2C#),
  11 => (LEAD =>16#1.44E#, TRAIL =>16#0.000086061892D03136F409DF019FBD4F3#),
  12 => (LEAD =>16#1.4BF#, TRAIL =>16#0.000DAD5362A271D4397AFEC42E20E0363#),
  13 => (LEAD =>16#1.534#, TRAIL =>16#0.0002B569D4F81DF0A83C49D86A63F4E67#),
  14 => (LEAD =>16#1.5AB#, TRAIL =>16#0.00007DD48542958C93015191EB345D88D#),
  15 => (LEAD =>16#1.624#, TRAIL =>16#0.0007EB03A5584B1F0FA06FD2DA42BB1CE#),
  16 => (LEAD =>16#1.6A0#, TRAIL =>16#0.0009E667F3BCC908B2FB1366EA957D3E3#),
  17 => (LEAD =>16#1.71F#, TRAIL =>16#0.00075E8EC5F73DD2370F2EF0ACD6CB434#),
  18 => (LEAD =>16#1.7A1#, TRAIL =>16#0.0001473EB0186D7D51023F6CDA1F5EF42#),
  19 => (LEAD =>16#1.825#, TRAIL =>16#0.00089994CCE128ACF88AFAB34A010F6AD#),
  20 => (LEAD =>16#1.8AC#, TRAIL =>16#0.000E5422AA0DB5BA7C55A192C9BB3E6ED#),
  21 => (LEAD =>16#1.937#, TRAIL =>16#0.00037B0CDC5E4F4501C3F2540A22D2FC4#),
  22 => (LEAD =>16#1.9C4#, TRAIL =>16#0.0009182A3F0901C7C46B071F2BE58DDAD#),
  23 => (LEAD =>16#1.A55#, TRAIL =>16#0.00003B23E255C8B424491CAF87BC8050A#),
  24 => (LEAD =>16#1.AE8#, TRAIL =>16#0.0009F995AD3AD5E8734D1773205A7FBC3#),
  25 => (LEAD =>16#1.B7F#, TRAIL =>16#0.00076F2FB5E46EAA7B081AB53C5354C88#),
  26 => (LEAD =>16#1.C19#, TRAIL =>16#0.0009BDD85529C2220CB12A091BA667944#),
  27 => (LEAD =>16#1.CB7#, TRAIL =>16#0.00020DCEF90691503CBD1E949DB761D95#),
  28 => (LEAD =>16#1.D58#, TRAIL =>16#0.00018DCFBA48725DA05AEB66E0DCA9F58#),
  29 => (LEAD =>16#1.DFC#, TRAIL =>16#0.00097337B9B5EB968CAC39ED291B7225A#),
  30 => (LEAD =>16#1.EA4#, TRAIL =>16#0.000AFA2A490D9858F73A18F5DB301F86D#),
  31 => (LEAD =>16#1.F50#, TRAIL =>16#0.000765B6E4540674F84B762862BAFF98F#) );


-- SCALE is a primitive function that returns a value scaled by a
-- specified power of two. The following implementation of
```

```
   -- SCALE is temporary, awaiting a package (possibly nonportable)
   -- of primitive functions implemented in the most effective way
   -- possible for a given machine.

   function SCALE ( X : LONGEST_FLOAT; M : LONGEST_INTEGER )
         return LONGEST_FLOAT is

      FACTOR     : LONGEST_FLOAT;
      POWERS     : LONGEST_FLOAT := 1.0;
      MULTIPLIER : LONGEST_FLOAT := 1.0;
      I, J       : LONGEST_INTEGER;

   begin

      if M > 0 then
        FACTOR := 2.0;
        I := M;
      else
        FACTOR := 0.5;
        I := -M;
      end if;
      POWERS := FACTOR;
      while I >= 2 loop
        J := I mod 2;
        I := I / 2;
        if J = 1 then
          MULTIPLIER := MULTIPLIER * POWERS;
        end if;
        POWERS := POWERS * POWERS;
      end loop;
      if M /= 0 then
         MULTIPLIER := MULTIPLIER * POWERS;
      end if;
      return ( MULTIPLIER*X );

   end SCALE;

   function EXP( X : FLOAT_TYPE ) return FLOAT_TYPE is separate;

   -- other functions to be added later

end GENERIC_ELEMENTARY_FUNCTIONS;


separate ( GENERIC_ELEMENTARY_FUNCTIONS )

function EXP( X : FLOAT_TYPE ) return FLOAT_TYPE is
```

```
   RESULT : FLOAT_TYPE;
   LOG2_BY_32_LEAD, LOG2_BY_32_TRAIL, F,
      Y, R1, R2, Q, TMP : LONGEST_FLOAT;
   TWO_TO_6 : constant := 64;
   TWO_TO_8 : constant := 256;
   LOG2_BY_32 : constant :=
      16#5.8B90B_FBE8E_7BCD5_E4F1D_9CC01_F97B5_7A079_A1933_94C#E-2;
   THIRTY_TWO_BY_LOG2 : constant :=
      16#2E.2A8EC_A5705_FC2EE_FA1FF_B41A4_74FA2_3AD5E#;
   LARGE_THRESHOLD : LONGEST_FLOAT :=
      2.0 * LONGEST_FLOAT(FLOAT_TYPE'SAFE_EMAX) * 6.931471806E-1;
   SMALL_THRESHOLD : LONGEST_FLOAT :=
      FLOAT_TYPE'BASE'EPSILON;

   I, N, N_1, N_2, M, J : LONGEST_INTEGER;

begin

   -- Step 1. Filter out exceptional cases.

   Y := LONGEST_FLOAT( X );
   if abs(Y) >= LARGE_THRESHOLD then
      raise ARGUMENT_ERROR;
   elsif abs(Y) <= SMALL_THRESHOLD then
      return( FLOAT_TYPE( 1.0 + Y ) );
   end if;


   -- Step 2. Reduce the argument.

   -- To perform argument reduction, we find the integer N such that
   --    X = N * log2/32 + remainder, |remainder| <= log2/64.
   -- N is defined by round-to-nearest-integer( X*32/log2 ) and
   -- remainder by X - N*log2/32. The calculation of N is
   -- straightforward whereas the computation of X - N*log2/32
   -- must be carried out carefully.
   -- log2/32 is so represented in two pieces that
   -- (1) log2/32 is known to extra precision, (2) the product
   -- of N and the leading piece is a model number and is hence
   -- calculated without error, and (3) the subtraction of the value
   -- obtained in (2) from X is a model number and is hence again obtained
   -- without error.

   -- The following case analysis chooses the appropriate
   -- representation of log2/32, depending on the number of
   -- digits in LONGEST_FLOAT.
```

```
case SYSTEM.MAX_DIGITS is

   when 6      =>

      LOG2_BY_32_LEAD  := 16#5.8B8#E-2;               --12 bits
      LOG2_BY_32_TRAIL := LOG2_BY_32 - 16#5.8B8#E-2;

   when 7..8   =>

      LOG2_BY_32_LEAD  := 16#5.8B9#E-2;               --15 bits
      LOG2_BY_32_TRAIL := LOG2_BY_32 - 16#5.8B9#E-2;

   when 9..11  =>

      LOG2_BY_32_LEAD  := 16#5.8B908#E-2;             --17  bits
      LOG2_BY_32_TRAIL := LOG2_BY_32 - 16#5.8B908#E-2;

   when 12..14 =>

      LOG2_BY_32_LEAD  := 16#5.8B90A#E-2;             --22  bits
      LOG2_BY_32_TRAIL := LOG2_BY_32 - 16#5.8B90A#E-2;

   when 15..19 =>

      LOG2_BY_32_LEAD  := 16#5.8B90B_F8#E-2;          --28  bits
      LOG2_BY_32_TRAIL := LOG2_BY_32 - 16#5.8B90B_F8#E-2;

   when 20..27 =>

      LOG2_BY_32_LEAD  := 16#5.8B90B_FBE8#E-2;        --37  bits
      LOG2_BY_32_TRAIL := LOG2_BY_32 - 16#5.8B90B_FBE8#E-2;

   when 28..33 =>

      LOG2_BY_32_LEAD  := 16#5.8B90B_FBE8E_7A#E-2;    --50  bits
      LOG2_BY_32_TRAIL := LOG2_BY_32 - 16#5.8B90B_FBE8E_7A#E-2;

   when others =>

      raise PROGRAM_ERROR;  -- assumption (1) is violated.

end case;

-- Perform argument reduction in LONGEST_FLOAT.

N := LONGEST_INTEGER( Y * THIRTY_TWO_BY_LOG2 );
if abs(N) >= TWO_TO_8 then
   N_2 := N mod TWO_TO_6;
   N_1 := N - N_2;
else
   N_2 := 0;
   N_1 := N;
end if;
TMP := LONGEST_FLOAT( N_1 ) * LOG2_BY_32_LEAD;
if abs( Y ) >= abs( TMP ) then
   R1 := Y - TMP;
else
   TMP := 0.5 * TMP;
   R1 := (Y - TMP) - TMP;
end if;
if N_2 /= 0 then
   R1 := R1 - LONGEST_FLOAT(N_2) * LOG2_BY_32_LEAD;
end if;
R2 := -LONGEST_FLOAT(N) * LOG2_BY_32_TRAIL;
J := N mod 32;
M := (N - J)/32;

-- Step 3. Approximation.

-- The following is the core approximation. We approximate
-- exp(R1+R2)-1 by a polynomial. The case analysis finds both
-- a suitable floating-point type (less expensive to use than
-- LONGEST_FLOAT) and an appropriate polynomial approximation
-- that will deliver a result accurate enough with respect to
-- FLOAT_TYPE'BASE'DIGITS. Note that the upper bounds of the
-- cases below (6, 15, 16, 18, 27, and 33) are attributes
-- of predefined floating types of common systems.

case FLOAT_TYPE'BASE'DIGITS is

   when 1..6 =>

      declare
         type WORKING_FLOAT is digits 6;
         R, POLY : WORKING_FLOAT;
      begin
         R := WORKING_FLOAT( R1 + R2 );
         POLY := R*R*( 5.00004_0481E-01 + R * 1.66667_6443E-01 );
         Q := R1 + ( R2 + LONGEST_FLOAT( POLY ) );
      end;

   when 7..15 =>

      declare
         type WORKING_FLOAT is digits
            (15+SYSTEM.MAX_DIGITS - abs(15-SYSTEM.MAX_DIGITS))/2;
         -- this is min( 15, SYSTEM.MAX_DIGITS )
         R, POLY : WORKING_FLOAT;
      begin
         R := WORKING_FLOAT( R1 + R2 );
         POLY := R*R*( 5.00000_00000_00000_08883E-01 +
                    R*( 1.66666_66666_52608_78863E-01 +
                    R*( 4.16666_66666_22607_95726E-02 +
                    R*( 8.33336_79843_42196_16221E-03 +
                    R*( 1.38889_49086_37771_99667E-03 )))));
         Q := R1 + ( R2 + LONGEST_FLOAT( POLY ) );
      end;

   when 16 =>

      declare
         type WORKING_FLOAT is digits
            (16+SYSTEM.MAX_DIGITS - abs(16-SYSTEM.MAX_DIGITS))/2;
         R, POLY : WORKING_FLOAT;
      begin
         R :=  WORKING_FLOAT( R1 + R2 );
         POLY := R*R*( 5.00000_00000_00000_08883E-01 +
                    R*( 1.66666_66666_52608_78863E-01 +
                    R*( 4.16666_66666_22607_95726E-02 +
                    R*( 8.33336_79843_42196_16221E-03 +
                    R*( 1.38889_49086_37771_99667E-03 )))));
         Q := R1 + ( R2 + LONGEST_FLOAT( POLY ) );
      end;

   when 17..18 =>

      declare
         type WORKING_FLOAT is digits
            (18+SYSTEM.MAX_DIGITS - abs(18-SYSTEM.MAX_DIGITS))/2;
         R, POLY : WORKING_FLOAT;
      begin
         R := WORKING_FLOAT( R1 + R2 );
         POLY := R*R*( 5.00000_00000_00000_07339E-01 +
                    R*( 1.66666_66666_66666_69177E-01 +
                    R*( 4.16666_66666_28680_32559E-02 +
                    R*( 8.33333_33332_52083_91118E-03 +
                    R*( 1.38889_44766_51246_30293E-03 +
                    R*( 1.98413_53190_32208_33704E-04 ))))));
         Q := R1 + ( R2 + LONGEST_FLOAT( POLY ) );
      end;

   when 19..27 =>

      declare
```

```
         type WORKING_FLOAT is digits
            (27+SYSTEM.MAX_DIGITS - abs(27-SYSTEM.MAX_DIGITS))/2;
         R, POLY : WORKING_FLOAT;
      begin
         R := WORKING_FLOAT( R1 + R2 );
         POLY := R*R*( 4.99999_99999_99999_99999_99636_21075E-01 +
                   R*( 1.66666_66666_66666_66666_66512_04136E-01 +
                   R*( 4.16666_66666_66666_69681_59325_03184E-02 +
                   R*( 8.33333_33333_33333_40906_33326_46233E-03 +
                   R*( 1.38888_88888_81124_92492_26093_01620E-03 +
                   R*( 1.98412_69841_13983_54303_59568_15543E-04 +
                   R*( 2.48016_66086_20855_39725_92760_56125E-05 +
                   R*( 2.75574_13983_51388_82843_29291_74995E-06
                       ))))))));
         Q := R1 + ( R2 + LONGEST_FLOAT( POLY ) );
      end;

   when 28..33 =>

      declare
         type WORKING_FLOAT is digits
            (33+SYSTEM.MAX_DIGITS - abs(33-SYSTEM.MAX_DIGITS))/2;
         R, POLY : WORKING_FLOAT;
      begin
         R := WORKING_FLOAT( R1 + R2 );
         POLY := R*R*( 5.0E-01 +
       R*( 1.66666_66666_66666_66666_66666_66668_18891E-01 +
       R*( 4.16666_66666_66666_66666_66666_66671_98062E-02 +
       R*( 8.33333_33333_33333_33333_33182_72433_96473E-03 +
       R*( 1.38888_88888_88888_88888_88860_77788_96115E-03 +
       R*( 1.98412_69841_26984_13216_98830_39302_820E-04 +
       R*( 2.48015_87301_58730_16549_32617_44006_810E-05 +
       R*( 2.75573_19223_90497_50521_23337_44713_411E-06 +
       R*( 2.75573_19223_90383_09381_24531_22474_208E-07 +
       R*( 2.50521_67036_89710_14700_24557_88635_351E-08 +
       R*( 2.08768_06002_87469_73970_46716_40247_597E-09
                  )))))))))));
         Q := R1 + ( R2 + LONGEST_FLOAT( POLY ) );
      end;

   when others =>

      raise PROGRAM_ERROR;  -- assumption (1) is violated.

end case;

-- This completes the core approximation.
```

```
   -- Step 4. Function value reconstruction.

   -- We now reconstruct the exponential of the input argument.
   -- The order of the computation below must be strictly observed.

   F := TWO_TO_J_BY_32( J, LEAD ) + TWO_TO_J_BY_32( J, TRAIL );

   case LONGEST_FLOAT'MACHINE_RADIX is
      when 2 =>

         Y := TWO_TO_J_BY_32( J, LEAD ) +
                    ( TWO_TO_J_BY_32( J, TRAIL ) + F*Q );
         RESULT := FLOAT_TYPE( SCALE( Y, M ) );

      when 16 =>

         Y := SCALE( TWO_TO_J_BY_32( J, LEAD ), M ) +
              SCALE( TWO_TO_J_BY_32( J, TRAIL ) + F*Q, M );
         RESULT := FLOAT_TYPE( Y );

      when others =>

         raise PROGRAM_ERROR;  -- assumption (1) is violated.

   end case;

   if RESULT /= 0.0 then
      return( RESULT );
   else
      raise ARGUMENT_ERROR;
   end if;

exception

   when NUMERIC_ERROR | CONSTRAINT_ERROR =>
      raise ARGUMENT_ERROR;
   -- This handling may be changed in the future. For one
   -- thing, overflowing the constraints of the base type
   -- of FLOAT_TYPE and overflowing the constraints of
   -- FLOAT_TYPE are indistinguishable in the way exception
   -- is handled now.

end EXP;
```

## References

[1] R. C. Agarwal et al., New scalar and vector elementary functions for the IBM System/370, *IBM Journal of Research and Development,* 30, no. 2, March 1986, pp. 126-144.

[2] W. S. Brown and S. I. Feldman, Environment parameters and basic functions for floating-point computation, *ACM Transactions on Mathematical Software,* 6, no. 4, December 1980, pp. 510-523.

[3] W. Cody and W. Waite, *Software Manual for the Elementary Functions,* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[4] J. Hart et al., *Computer Approximations,* John Wiley and Sons, N.Y., 1968.

[5] *Elementary Math Library, Programming RPQ P81005, Program Number 5799-BTB, Program Reference and Operations Manual,* August 1984, SH20-2230-1.

[6] J. Kok, Proposal for standard mathematical packages in Ada, *Report NM-R8718,* Center for Mathematics and Computer Science, Amsterdam, The Netherlands, November 1987.

[7] R. F. Mathis, Elementary Functions Package for Ada, *Proceedings for ACM SIGAda International Conference*, Boston, Massachusetts, December 1987.

## Distribution for ANL-88-3

**Internal:**

J. M. Beumer (3)
W. J. Cody
K. W. Dritz
F. Y. Fradin
H. G. Kaper
A. B. Krisciunas
G. W. Pieper (50)
B. T. Smith
P. Tang (30)

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (3)

**External:**

DOE-TIC, for distribution per UC-32 (112)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. L. Bona, Pennsylvania State University
T. L. Brown, University of Illinois, Urbana
P. Concus, Lawrence Berkeley Laboratory
S. Gerhart, Micro Electronics and Computer Technology Corp., Austin, TX
H. B. Keller, California Institute of Technology
J. A. Nohel, University of Wisconsin, Madison
M. J. O'Donnell, University of Chicago
Capt. David Hart, Office of the Secretary of Defense
Capt. Stephen Johnson, Office of the Secretary of Defense
G. Michael, Lawrence Livermore Laboratory
Lt. Col. Jon Rindt, Office of the Secretary of Defense
Lt. Col. Peter Sowa, Office of the Secreteary of Defense